Introduction

- What are embedded systems?
- Challenges in embedded computing system design.
- Design methodologies.

Definition

- Embedded system: any device that includes a programmable computer but is not itself a general-purpose computer.
- Take advantage of application characteristics to optimize the design:
  - don't need all the general-purpose bells and whistles.

Definition, cont’d.

- Embedded system: any device that includes a programmable computer but is not itself a general-purpose computer.
- Take advantage of application characteristics to optimize the design:
  - don't need all the general-purpose bells and whistles.

Embedding a computer

Examples

- Personal digital assistant (PDA).
- Printer.
- Cell phone.
- Automobile: engine, brakes, dash, etc.
- Television.
- Household appliances.
- PC keyboard (scans keys).

Early history

- Late 1940’s: MIT Whirlwind computer was designed for real-time operations.
  - Originally designed to control an aircraft simulator.
  - First microprocessor was Intel 4004 in early 1970’s.
- HP-35 calculator used several chips to implement a microprocessor in 1972.

Early history, cont’d.

- Automobiles used microprocessor-based engine controllers starting in 1970’s.
  - Control fuel/air mixture, engine timing, etc.
  - Multiple modes of operation: warm-up, cruise, hill climbing, etc.
  - Provides lower emissions, better fuel efficiency.
**Microprocessor varieties**

- **Microcontroller**: includes I/O devices, on-board memory.
- **Digital signal processor (DSP)**: microprocessor optimized for digital signal processing.
- Typical embedded word sizes: 8-bit, 16-bit, 32-bit.

**Application examples**

- Simple control: front panel of microwave oven, etc.
- Canon EOS 3 has three microprocessors.
  - 32-bit RISC CPU runs autofocus and eye control systems.
- Analog TV: channel selection, etc.
- Digital TV: programmable CPUs + hardwired logic.

**Automotive embedded systems**

- Today’s high-end automobile may have 100 microprocessors:
  - 4-bit microcontroller checks seat belt;
  - microcontrollers run dashboard devices;
  - 16/32-bit microprocessor controls engine.

**BMW 850i brake and stability control system**

- **Anti-lock brake system (ABS)**: pumps brakes to reduce skidding.
- **Automatic stability control (ASC+T)**: controls engine to improve stability.
- ABS and ASC+T communicate.
  - ABS was introduced first—needed to interface to existing ABS module.

**BMW 850i, cont’d.**

**Characteristics of embedded systems**

- Sophisticated functionality.
- Real-time operation.
- Low manufacturing cost.
- Low power.
- Designed to tight deadlines by small teams.
**Functional complexity**

- Often have to run sophisticated algorithms or multiple algorithms.
  - Cell phone, laser printer.
- Often provide sophisticated user interfaces.

**Real-time operation**

- Must finish operations by deadlines.
  - **Hard real time:** missing deadline causes failure.
  - **Soft real time:** missing deadline results in degraded performance.
- Many systems are multi-rate: must handle operations at widely varying rates.

**Non-functional requirements**

- Many embedded systems are mass-market items that must have low manufacturing costs.
  - Limited memory, microprocessor power, etc.
- Power consumption is critical in battery-powered devices.
  - Excessive power consumption increases system cost even in wall-powered devices.

**Design teams**

- Often designed by a small team of designers.
- Often must meet tight deadlines.
  - 6 month market window is common.
  - Can't miss back-to-school window for calculator.

**Why use microprocessors?**

- Alternatives: field-programmable gate arrays (FPGAs), custom logic, etc.
- Microprocessors are often very efficient: can use same logic to perform many different functions.
- Microprocessors simplify the design of families of products.

**The performance paradox**

- Microprocessors use much more logic to implement a function than does custom logic.
- But microprocessors are often at least as fast:
  - heavily pipelined;
  - large design teams;
  - aggressive VLSI technology.
Power

- Custom logic is a clear winner for low power devices.
- Modern microprocessors offer features to help control power consumption.
- Software design techniques can help reduce power consumption.

Challenges in embedded system design

- How much hardware do we need?
  - How big is the CPU? Memory?
- How do we meet our deadlines?
  - Faster hardware or cleverer software?
- How do we minimize power?
  - Turn off unnecessary logic? Reduce memory accesses?

Challenges, etc.

- Does it really work?
  - Is the specification correct?
  - Does the implementation meet the spec?
  - How do we test for real-time characteristics?
  - How do we test on real data?
- How do we work on the system?
  - Observability, controllability?
  - What is our development platform?

Design methodologies

- A procedure for designing a system.
- Understanding your methodology helps you ensure you didn't skip anything.
- Compilers, software engineering tools, computer-aided design (CAD) tools, etc., can be used to:
  - help automate methodology steps;
  - keep track of the methodology itself.

Design goals

- Performance.
  - Overall speed, deadlines.
- Functionality and user interface.
- Manufacturing cost.
- Power consumption.
- Other requirements (physical size, etc.)

Levels of abstraction

- requirements
- specification
- architecture
- component design
- system design
- integration
Top-down vs. bottom-up

- Top-down design:
  - start from most abstract description;
  - work to most detailed.
- Bottom-up design:
  - work from small components to big system.
- Real design uses both techniques.

Stepwise refinement

- At each level of abstraction, we must:
  - analyze the design to determine characteristics of the current state of the design;
  - refine the design to add detail.

Requirements

- Plain language description of what the user wants and expects to get.
- May be developed in several ways:
  - talking directly to customers;
  - talking to marketing representatives;
  - providing prototypes to users for comment.

Functional vs. non-functional requirements

- Functional requirements:
  - output as a function of input.
- Non-functional requirements:
  - time required to compute output;
  - size, weight, etc.;
  - power consumption;
  - reliability;
  - etc.

Our requirements form

- name
- purpose
- inputs
- outputs
- functions
- performance
- manufacturing cost
- power
- physical size/weight

Example: GPS moving map requirements

- Moving map obtains position from GPS, paints map from local database.
GPS moving map needs

- **Functionality:** For automotive use. Show major roads and landmarks.
- **User interface:** At least 400 x 600 pixel screen. Three buttons max. Pop-up menu.
- **Performance:** Map should scroll smoothly. No more than 1 sec power-up. Lock onto GPS within 15 seconds.
- **Cost:** $500 street price = approx. $100 cost of goods sold.

GPS moving map needs, cont’d.

- **Physical size/weight:** Should fit in hand.
- **Power consumption:** Should run for 8 hours on four AA batteries.

GPS moving map requirements form

<table>
<thead>
<tr>
<th>name</th>
<th>GPS moving map requirements form</th>
</tr>
</thead>
<tbody>
<tr>
<td>purpose</td>
<td>consumer-grade</td>
</tr>
<tr>
<td>inputs</td>
<td>moving map for driving</td>
</tr>
<tr>
<td></td>
<td>power button, two</td>
</tr>
<tr>
<td></td>
<td>control buttons</td>
</tr>
<tr>
<td>outputs</td>
<td>back-lit LCD 400 x 600</td>
</tr>
<tr>
<td>functions</td>
<td>5-receiver GPS; three resolutions; displays current latitude</td>
</tr>
<tr>
<td>performance</td>
<td>updates screen within 0.25 sec of movement</td>
</tr>
<tr>
<td>manufacturing cost</td>
<td>$100 cost of goods-sold</td>
</tr>
<tr>
<td>power</td>
<td>100 mW</td>
</tr>
<tr>
<td>physical size/weight</td>
<td>no more than 2: X 6:, 12 oz.</td>
</tr>
</tbody>
</table>

Specification

- A more precise description of the system:
  - should not imply a particular architecture;
  - provides input to architecture design process.
- May include functional and non-functional elements.
- May be executable or may be in mathematical form for proofs.

GPS specification

- **Should include:**
  - What is received from GPS;
  - map data;
  - user interface;
  - operations required to satisfy user requests;
  - background operations needed to keep the system running.

Architecture design

- What major components go satisfying the specification?
  - Hardware components:
    - CPUs, peripherals, etc.
  - Software components:
    - major programs and their operations.
  - Must take into account functional and non-functional specifications.
GPS moving map block diagram

GPS receiver | search engine | renderer | display
-----------|-------------|---------|--------
database   |             | user interface |       

GPS moving map hardware architecture

display | frame buffer | CPU
--------|-------------|------
memory  | GPS receiver | panel I/O

GPS moving map software architecture

position | database search | renderer | pixels
---------|----------------|---------|-------
user interface |            | timer |       

Designing hardware and software components

- Must spend time architecting the system before you start coding.
- Some components are ready-made, some can be modified from existing designs, others must be designed from scratch.

System integration

- Put together the components.
  - Many bugs appear only at this stage.
  - Have a plan for integrating components to uncover bugs quickly, test as much functionality as early as possible.

Summary

- Embedded computers are all around us.
  - Many systems have complex embedded hardware and software.
- Embedded systems pose many design challenges: design time, deadlines, power, etc.
- Design methodologies help us manage the design process.
Introduction

- Object-oriented design.
- Unified Modeling Language (UML).

System modeling

- Need languages to describe systems:
  - useful across several levels of abstraction;
  - understandable within and between organizations.
- Block diagrams are a start, but don’t cover everything.

Object-oriented design

- Object-oriented (OO) design: A generalization of object-oriented programming.
- Object = state + methods.
  - State provides each object with its own identity.
  - Methods provide an abstract interface to the object.

Objects and classes

- Class: object type.
- Class defines the object’s state elements but state values may change over time.
- Class defines the methods used to interact with all objects of that type.
  - Each object has its own state.

OO design principles

- Some objects will closely correspond to real-world objects.
- Some objects may be useful only for description or implementation.
- Objects provide interfaces to read/write state, hiding the object’s implementation from the rest of the system.

UML

- Developed by Booch et al.
- Goals:
  - object-oriented;
  - visual;
  - useful at many levels of abstraction;
  - usable for all aspects of design.
The class interface

- The operations provide the abstract interface between the class’s implementation and other classes.
- Operations may have arguments, return values.
- An operation can examine and/or modify the object’s state.

Choose your interface properly

- If the interface is too small/specialized:
  - object is hard to use for even one application;
  - even harder to reuse.
- If the interface is too large:
  - class becomes too cumbersome for designers to understand;
  - implementation may be too slow;
  - spec and implementation are probably buggy.

Relationships between objects and classes

- Association: objects communicate but one does not own the other.
- Aggregation: a complex object is made of several smaller objects.
- Composition: aggregation in which owner does not allow access to its components.
- Generalization: define one class in terms of another.

Class derivation

- May want to define one class in terms of another.
  - Derived class inherits attributes, operations of base class.
Class derivation example

- **Base class**
  - Display
  - pixels
  - elements
  - menu_items

- **Derived class**
  - BW_display
  - Color_map_display

Multiple inheritance

- **Base classes**
  - Speaker
  - Display

Links and associations

- **Link**: describes relationships between objects.
- **Association**: describes relationship between classes.

Link example

- **Link** defines the *contains* relationship:

  - **Message**
    - msg := msg1
    - length = 1102
  - **Message set**
    - count := 2

Association example

- **# contained messages**
  - message
  - msg: ADPCM_stream
  - length: integer

- **# containing message sets**
  - message set
  - contains

Stereotypes

- **Stereotype**: recurring combination of elements in an object or class.
- **Example**:
  - «foo>»
Behavioral description

Several ways to describe behavior:
- internal view;
- external view.

State machines

Event-driven state machines

Behavioral descriptions are written as event-driven state machines.
- Machine changes state when receiving an input.
- An event may come from inside or outside of the system.

Types of events

- Signal: asynchronous event.
- Call: synchronized communication.
- Timer: activated by time.

Signal event

```xml
<signal>
  mouse_click
  leftorright: button
  x, y: position
</signal>
```

declaration

```xml
mouse_click(x,y,button)
```
event description

Call event

```
c
```
```
d
```
draw_box(10,5,3,blue)```
Timer event

tm(time-value)

c → f

Example state machine

start

mouse_click(x,y,button)
find_region(region)

found

region = menu/
which_menu(i)
call_menu(i)

got menu item

called menu item

input/output

region = drawing/
find_object(objid)
highlight(objid)

found

object

highlighted

finish

Sequence diagram

t Shows sequence of operations over time.

r Relates behaviors of multiple objects.

Sequence diagram example

m: Mouse
d: Display
u: Menu

mouse_click(x,y,button)

which_menu(x,y,i)
call_menu(i)

time

Summary

r Object-oriented design helps us organize a design.

r UML is a transportable system design language.

r Provides structural and behavioral description primitives.

Introduction

r Example: model train controller.
**Requirements**

- Console can control 8 trains on 1 track.
- Throttle has at least 63 levels.
- Inertia control adjusts responsiveness with at least 8 levels.
- Emergency stop button.
- Error detection scheme on messages.

**Conceptual specification**

- Before we create a detailed specification, we will make an initial, simplified specification.
  - Gives us practice in specification and UML.
  - Good idea in general to identify potential problems before investing too much effort in detail.
Typical control sequence:

- `console`
- `set-inertia`
- `set-speed`
- `estop`
- `train_rev`

Message classes:

- `command`
- `set-speed`
- `set-inertia`
- `estop`

Roles of message classes:

- Implemented message classes derived from message class.
  - Attributes and operations will be filled in for detailed specification.
- Implemented message classes specify message type by their class.
  - May have to add type as parameter to data structure in implementation.

Subsystem collaboration diagram:

Shows relationship between console and receiver (ignores role of track):

```
1..n: command
```

System structure modeling:

- Some classes define non-computer components.
  - Denote by *name.
- Choose important systems at this point to show basic relationships.

Major subsystem roles:

- **Console:**
  - read state of front panel;
  - format messages;
  - transmit messages.
- **Train:**
  - receive message;
  - interpret message;
  - control the train.
Console system classes

1. console
   - panel
     - formatter
     - transmitter
   - receiver*
   - sender*

Train system classes

1. train set
   - train
     - controller
     - motor
     - interface
   - receiver
   - detector*
   - pulser*

Console class roles

- panel: describes analog knobs and interface hardware.
- formatter: turns knob settings into bit streams.
- transmitter: sends data on track.

Train class roles

- receiver: digitizes signal from track.
- controller: interprets received commands and makes control decisions.
- motor interface: generates signals required by motor.

Detailed specification

- We can now fill in the details of the conceptual specification:
  - more classes;
  - behaviors.
- Sketching out the spec first helps us understand the basic relationships in the system.

Train speed control

- Motor controlled by pulse width modulation:
**Console physical object classes**

- **knobs**:
  - train-knob: integer
  - speed-knob: integer
  - inertia-knob: unsigned-integer
  - emergency-stop: boolean

- **pulser**:
  - pulse-width: unsigned-integer
  - direction: boolean

- **sender**:
  - send-bit()

- **detector**:
  - read-bit(): integer

**Panel and motor interface classes**

- **panel**:
  - train-number(): integer
  - speed(): integer
  - inertia(): integer
  - estop(): boolean
  - new-settings()

- **motor-interface**:
  - speed: integer

**Class descriptions**

- Panel class defines the controls.
  - new-settings() behavior reads the controls.
- Motor-interface class defines the motor speed held as state.

**Class descriptions**

- Transmitter class has one behavior for each type of message sent.
- Receiver function provides methods to:
  - detect a new message;
  - determine its type;
  - read its parameters (estop has no parameters).

**Formatter class**

- Formatter
  - current-train: integer
  - current-speed[ntrains]: integer
  - current-inertia[ntrains]: unsigned-integer
  - current-estop[ntrains]: boolean
  - send-command()
  - panel-active() : boolean
  - operate()
**Formatter class description**

- Formatter class holds state for each train, setting for current train.
- The operate() operation performs the basic formatting task.

**Control input cases**

- Use a soft panel to show current panel settings for each train.
- Changing train number:
  - must change soft panel settings to reflect current train’s speed, etc.
- Controlling throttle/inertia/estop:
  - read panel, check for changes, perform command.

**Formatter operate behavior**

```
Formatter operate

idle

update-panel()

other

send-command()
```

**Controller class**

```
controller

controller: controller

current-train: integer

current-speed: integer

current-direction: boolean

current-inertia: unsigned-integer

operate()

issue-command()
```

**Control input sequence diagram**

```
knobs:
  - change in control settings
  - change in train number
  - set-knobs

panel:
  - read panel
  - read panel
  - panel-active

formatter:
  - send-command
  - send-speed
  - send-inertia
  - send-estop

transmitter:
  - send-speed
  - send-inertia
  - send-estop

change in speed/inertia/estop:

change in train number:

out
```

**Panel-active behavior**

```
panel*: read-train()

T current-train = train-knob

update-screen

changed = true

F

panel*: read-speed()

T current-speed = throttle

changed = true

F
```

© 2000 Morgan Kaufman

Overheads for Computers as Components
Setting the speed

- Don’t want to change speed instantaneously.
- Controller should change speed gradually by sending several commands.

Controller operate behavior

- wait for a command from receiver
- receive-command()
- issue-command()

Sequence diagram for set-speed command

Refined command classes

Summary

- Separate specification and programming.
  - Small mistakes are easier to fix in the spec.
  - Big mistakes in programming cost a lot of time.
- You can’t completely separate specification and architecture.
  - Make a few tasteful assumptions.